Abstract Interface Behavior of Object-Oriented Languages with Monitors

Erika Ábrahám[1], Andreas Grüner[2], and Martin Steffen[2]

[1] Albert-Ludwigs-University Freiburg, Germany
[2] Christian-Albrechts-University Kiel, Germany

Abstract. We characterize the observable behavior of multi-threaded, object-oriented programs with *re-entrant monitors*. The observable uncertainty at the interface is captured by may- and must-approximations for potential resp. necessary lock ownership. The concepts are formalized in an object calculus. We show the soundness of the abstractions.

Keywords: oo languages, formal semantics, thread-based concurrency, monitors, open systems, observable behavior

1 Introduction

The behavior of an open system or component can be described by sequences of component-environment interactions. Even if the environment is absent, it must be assumed that the component together with the (abstracted) environment gives a well-formed program adhering to the syntactical and the context-sensitive restrictions of the language at hand. Technically, for an exact representation of the interface behavior, the semantics of the open program needs to be formulated under *assumptions* about the environment, capturing those restrictions. The resulting assumption-commitment framework gives insight to the semantical nature of the language at hand. Furthermore, an independent characterization of possible interface behavior with environment and component abstracted can be seen as a trace logic under the most general assumptions, namely conformance to the inherent restrictions of the language and its semantics.

This paper deals primarily with the following features, which correspond to those of modern class-based object-oriented languages like *Java* [8] or $C^{\#}$ [6] and which are notoriously hard to capture:

- *types and classes:* the languages are statically typed, and only well-typed programs are considered.
- *references:* each object carries a unique *identity.* New objects are dynamically allocated on the heap.
- *concurrency:* the mentioned languages feature concurrency based on *threads* (as opposed to processes or active objects).
- *monitor synchronization:* objects can play the role of monitors [9][5], guaranteeing that synchronized methods are executed mutually exclusive. Recursion —direct or indirect— via method call requires *re-entrant* monitors.

We investigate these issues in a class-based, multi-threaded calculus with monitors. The interface behavior is formulated in an assumption-commitment framework and based on three orthogonal abstractions:

- a static abstraction, i.e., the type system;
- an abstraction of the stacks of recursive method invocations, representing the recursive nature of method calls in a multi-threaded setting;
- finally as the main contribution, an abstraction of *lock ownership*.

The contribution of this paper over our previous work in this field (e.g., [2] dealing with deterministic, single-threaded programs, or [4] considering thread classes) is to capture re-entrant monitor behavior. In comparison with the mentioned work, the setting here is simpler in one respect: We disallow instantiation across the interface here; of course, instantiation as such is supported, only not across the boundary between component and environment.

Incorporating monitors into the formal calculus is not only pragmatically motivated —after all, *Java* and similar languages offer monitor synchronization— but also semantically interesting, because the observable equivalences induced by a language offering synchronized methods and one without are incomparable.

Overview Section 2 contains syntax and operational semantics of the calculus. Section 3 contains an independent characterization of the interface behavior of an open system, especially capturing the effects of lock ownership. Furthermore, it contains the basic soundness results of the abstractions. Section 4 concludes with related and future work. For a full account of the operational semantics and the type system, we refer to the technical report [3].

2 A multi-threaded calculus with classes

This section presents the calculus, which is based on a multi-threaded object calculus, similar to the one presented in [7] and in particular [10].

2.1 Syntax

The abstract syntax is given in Table 1. A program is given by a collection of classes where a class $c[\![O]\!]$ carries a name c and defines its methods and fields. We generally use o and its syntactic variants as names for objects, c for classes, and n for thread names and when being unspecific. An object $o[c, F, n]$ keeps a reference to the class c it instantiates, stores the current value of the fields or instance variables, and maintains a *lock* n, referring to the name of the thread holding the lock. The special name $\bot_{thread}$ (which is not a value) denotes that the lock is free. Immediately after instantiation, all fields carry the undefined reference $\bot_c$, where c is the (return) type of the field, and the lock is free. A method $\varsigma(self{:}c).\lambda(\vec{x}{:}\vec{T}).t$ provides the method body abstracted over the ς-bound "self" parameter and the formal parameters of the method [1]. We

$C ::= \mathbf{0} \mid C \parallel C \mid \nu(n{:}T).C \mid n[\![O]\!] \mid n[n, F, n] \mid n\langle t\rangle$	program
$O ::= F, M$	object
$M ::= l^u = m, \ldots, l^u = m, l^s = m, \ldots, l^s = m$	method suite
$F ::= l^u = f, \ldots, l^u = f$	fields
$m ::= \varsigma(n{:}T).\lambda(x{:}T, \ldots, x{:}T).t$	method
$f ::= \varsigma(n{:}T).\lambda().v \mid \varsigma(n{:}T).\lambda().\bot_n$	field
$t ::= v \mid \mathit{stop} \mid \mathit{let}\, x{:}T = e\ \mathit{in}\, t$	thread
$e ::= t \mid \mathsf{if}\ v = v\ \mathsf{then}\ e\ \mathsf{else}\ e \mid \mathsf{if}\ \mathit{undef}(v.l)\ \mathsf{then}\ e\ \mathsf{else}\ e$	expr.
$\quad\mid\ v.l(v, \ldots, v) \mid v.l := v \mid \mathit{currentthread}$	
$\quad\mid\ \mathit{new}\, n \mid \mathit{new}\langle t\rangle$	
$v ::= x \mid n$	values

Table 1. Abstract syntax

distinguish between synchronized and un-synchronized methods conventionally by superscripts l^s resp. l^u, and just l when unspecific. Besides objects and classes, the dynamic configuration of a program contains threads $n\langle t\rangle$ as active entities.

A thread is basically either a value or a *let*-construct, which is used for local declarations and sequencing[3] of expressions, notably method calls (written $v.l(\vec{v})$), the creation of new objects $\mathit{new}\, c$ where c is a (component) class name, and *thread creation* $\mathit{new}\langle t\rangle$. We use f for instance variables or fields, $l = v$ for field variable declaration. Field access is written as $v.l$, and field update as $v'.l := v$. Apart from disallowing instantiation cross the interface between component and environment, we impose the following two restrictions on the language: firstly, we disallow direct access (read or write) to fields across object boundaries. Secondly, we forbid that any occurrence of thread creation $\mathit{new}\langle t\rangle$ contains a self-parameter, i.e., a name occurring bound by ς. The reason is that a new thread must start its life "outside" any monitor.

The available types include *thread* as the type of threads. Furthermore, objects are typed by the name of their class. As auxiliary types we have $T_1 \times \ldots \times T_k \to T$ as the type of methods, and furthermore $[l_1{:}U_1, \ldots, l_k{:}U_k]$ as the type or interface of unnamed objects, and $[\![l_1{:}U_1, \ldots, l_k{:}U_k]\!]$ as the type for classes. For brevity, we omit the definition of the type system, as it is straightforward.

2.2 Operational semantics

The operational semantics is given in two stages, component internal steps and external ones, the latter describe the interaction at the interface. The external steps are defined in reference to assumption and commitment contexts. The static part of the contexts corresponds to the static type system (cf. again [4]) and takes care that, e.g., only well-typed values are received from the environment.

[3] Sequential composition $t_1; t_2$ of two threads stands for $\mathit{let}\, x{:}T = t_1\ \mathit{in}\, t_2$, where x does not occur free in t_2.

2.2.1 Internal steps Table 2 contains a few typical internal reduction steps (the ones for conditionals, sequencing via let, thread creation, etc., are straightforward), distinguishing between confluent steps, written $\rightsquigarrow$, and other internal transitions, written $\xrightarrow{\tau}$. The $\textsc{Call}_i$-rules treat internal method calls, i.e., a call to an object contained in the configuration, where for synchronized methods, $\textsc{Call}^s_{i_1}$ takes the free lock and adds a release-action at the end of the method body. Rule $\textsc{Call}^s_{i_2}$ describes re-entrant calls. In the call-steps, $M.l(o)(\vec{v})$ resp. $O.l(o)(\vec{v})$ stands for $t[o/s][\vec{v}/\vec{x}]$, when the method suite $[M]$ resp. the object implementation $[O]$ equals $[\ldots, l = \varsigma(s{:}T).\lambda(\vec{x}{:}\vec{T}).t, \ldots]$. The rule $\textsc{Call}^u_i$ additional deals with field access. Note that the step is a $\xrightarrow{\tau}$-step, not a confluent one. The above reduction relations are used modulo *structural congruence,* which captures the algebraic properties of parallel composition and the hiding operator.

2.2.2 External steps A component exchanges information with the environment via *calls* and *returns.* In the labels, n is the thread that issues the call or returns from the call. Note that there are no separate external labels for object instantiation; we have forbidden cross-border instantiation. Given a label $\nu(\Xi).\gamma'$ where Ξ is a name context, i.e., a sequence of single $\nu(n{:}T)$ bindings and where γ' does not contain any binders; we call γ' the *core* of the label. Note that for incoming labels, Ξ contains only bindings to environment objects and at most one thread name; dually for outgoing communication. Given a label γ, we refer with $\lfloor\gamma\rfloor$ to its core. Furthermore, $thread(\gamma)$ denotes the thread of the label. The definitions are used analogously for send and receive labels. We write shortly γ_c for call and γ_r for return labels.

$$\gamma ::= n\langle call\ o.l(\vec{v})\rangle \mid n\langle return(v)\rangle \mid \nu(n{:}T).\gamma \qquad \text{basic labels}$$
$$a ::= \gamma? \mid \gamma! \qquad \text{receive and send labels}$$

$$c[\![F, M]\!] \parallel n\langle let\, x{:}c = new\, c\, in\, t\rangle \rightsquigarrow$$
$$c[\![F, M]\!] \parallel \nu(o{:}c).(o[c, F, \bot_{thread}] \parallel n\langle let\, x{:}c = o\, in\, t\rangle) \qquad \textsc{NewO}_i$$
$$c[\![F, M]\!] \parallel o[c, F', n'] \parallel n\langle let\, x{:}T = o.l^u(\vec{v})\, in\, t\rangle \xrightarrow{\tau}$$
$$c[\![F, M]\!] \parallel o[c, F', n'] \parallel n\langle let\, x{:}T = O.l^u(o)(\vec{v})\, in\, t\rangle \qquad \textsc{Call}^u_i$$
$$c[\![F, M]\!] \parallel o[c, F', \bot_{thread}] \parallel n\langle let\, x{:}T = o.l^s(\vec{v})\, in\, t\rangle \rightsquigarrow$$
$$c[\![F, M]\!] \parallel o[c, F', n] \parallel n\langle let\, x{:}T = M.l^s(o)(\vec{v})\, in\, release(o); t\rangle \qquad \textsc{Call}^s_{i_1}$$
$$c[\![F, M]\!] \parallel o[c, F', n] \parallel n\langle let\, x{:}T = o.l^s(\vec{v})\, in\, t\rangle \rightsquigarrow$$
$$c[\![F, M]\!] \parallel o[c, F', n] \parallel n\langle let\, x{:}T = M.l^s(o)(\vec{v})\, in\, t\rangle \qquad \textsc{Call}^s_{i_2}$$
$$o[c, F, n] \parallel n\langle let\, x{:}T = release(o)\, in\, t\rangle \xrightarrow{\tau} o[c, F, \bot_{thread}] \parallel n\langle t\rangle \qquad \textsc{Release}$$

Table 2. Internal steps

The external semantics is formalized as labeled transitions between judgments of the form $\Delta, \Sigma \vdash C : \Theta, \Sigma$, where Δ, Σ represent the *assumptions* about the environment of the component C and Θ, Σ the *commitments*. The assumptions require the existence (plus static typing information) of *named entities* in the environment. The semantics maintains as invariant that the assumption and commitment contexts are disjoint concerning object and class names, whereas a thread name occurs as assumption iff. it is mentioned in the commitments. By convention, the contexts Σ (and their alphabetic variants) contain exactly all bindings for thread names. This means, as invariant we maintain for all judgments $\Delta, \Sigma \vdash C : \Theta, \Sigma$ that Δ, Σ, and Θ are pairwise disjoint. The operational semantics is formulated as transitions between typed judgments $\Delta, \Sigma \vdash C : \Theta, \Sigma \xrightarrow{a} \acute{\Delta}, \acute{\Sigma} \vdash \acute{C} : \acute{\Theta}, \acute{\Sigma}$.

Notation 1 *We abbreviate the triple of name contexts Δ, Σ, Θ as Ξ. Furthermore we understand $\acute{\Delta}, \acute{\Sigma}, \acute{\Theta}$ as $\acute{\Xi}$, etc.*

The open semantics checks the *static* assumptions, i.e., whether at most the names actually occurring in the core of the label are mentioned in the ν-binders of the label, and whether the transmitted values are of the correct types. We write $\Xi \vdash a : T$ for that check, where T is type of the expression in the program that gives rise to the label. We omit the exact definition here (see [3]).

Besides *checking* whether the assumptions are met before a transition, the contexts are *updated* by a transition step, reflecting the change of knowledge.

Definition 1 (Context update). *For an incoming label $a = \nu(\Xi')\lfloor a \rfloor$ where n is a thread name s.t. $\Xi' \vdash n$, we define $\acute{\Xi}$ as:*

$$\acute{\Theta} = \Theta + \Theta',\ \acute{\Delta} = \Delta + (\Delta', \odot_n),\ \text{and}\ \acute{\Sigma} = \Sigma + \Sigma'.$$

In case $\Xi' \not\vdash n$, the summand $\odot_n$ is omitted. We write $\Xi + a$ for the update of Ξ. The update for outgoing communication is defined dually.

The operational rules of Table 3 use two additional expressions *blocks* and *returns v*. The three CallI-rules deal with incoming calls. For all three cases, the contexts are *updated* to $\acute{\Xi}$ to include the information concerning new objects and threads. Furthermore, it is *checked* whether the label is type-correct and that the step is possible according to the (updated) assumptions $\acute{\Xi}$. In the rules, $fn(\lfloor a \rfloor)$ refers to the free names of $\lfloor a \rfloor$ (which equal $names(\lfloor a \rfloor)$). Outgoing calls are dealt with in rule CallO. To distinguish the situation from component-internal calls, the receiver must be part of the environment, expressed by $\Delta \vdash o_r$. Starting with a well-typed component, there is no need in re-checking now that only values of appropriate types are handed out, as the operational steps preserve well-typedness ("subject reduction"). In addition to the rules of Table 3, there are similar ones for communication via returns (cf. [3]).

Note that the steps of Table 3 are independent of *lock* manipulations, e.g., an incoming call, which hands over the message via one of the CallI-rules does not attempt to obtain the lock; this is done by the internal steps from Table 2. This *decouples* the responsibilities of component and environment in the spirit of the

$$\frac{a = \nu(\Xi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad t_{blocked} = let\,x':T' = blocks\,in\,t \quad \acute{\Xi} = \Xi + a \quad \acute{\Xi} \vdash \lfloor a \rfloor : T}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle t_{blocked}\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\Xi_1).(C \parallel n\langle let\,x{:}T = o_r.l(\vec{v})\ in\ returns\ x; t_{blocked}\rangle)}\ \text{CALLI}_1$$

$$\frac{a = \nu(\Xi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad \Delta \vdash \odot_n \quad \acute{\Xi} = \Xi + a \quad \acute{\Xi} \vdash \lfloor a \rfloor : T}{\Xi \vdash C \parallel n\langle stop\rangle \xrightarrow{a} \acute{\Xi} \vdash C \parallel n\langle let\,x{:}T = o_r.l(\vec{v})\ in\ returns\ x; stop\rangle}\ \text{CALLI}_2$$

$$\frac{a = \nu(\Xi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad \Xi' \vdash n : thread \quad \acute{\Xi} = \Xi + a \quad \acute{\Xi} \vdash \lfloor a \rfloor : T}{\Xi \vdash C \xrightarrow{a} \acute{\Xi} \vdash C \parallel n\langle let\,x{:}T = o_r.l(\vec{v})\ in\ returns\ x; stop\rangle}\ \text{CALLI}_3$$

$$\frac{a = \nu(\Xi').\ n\langle call\ o_r.l(\vec{v})\rangle! \quad \Xi' = fn(\lfloor a \rfloor) \cap \Xi \quad \acute{\Xi}_1 = \Xi_1 \setminus \Xi' \quad \acute{\Delta} \vdash o_r \quad \acute{\Xi} = \Xi + a}{\Xi \vdash \nu(\Xi_1).(C \parallel n\langle let\,x{:}T = o_r.l(\vec{v})\ in\ t\rangle) \xrightarrow{a} \acute{\Xi} \vdash \nu(\acute{\Xi}_1).(C \parallel n\langle let\,x{:}T = blocks\,in\,t\rangle)}\ \text{CALLO}$$

Table 3. External steps

assumption/commitment set-up. Whether an incoming call can be sent by the environment depends *only* on the past interface interaction and the environment, *but not* on an internal state of the component!

3 Interface behavior

Next we characterize the possible ("legal") interface behavior as interaction traces between component and environment. The calls and returns of each thread must be "parenthetic", i.e., each return must have a prior matching call, and we must take into account whether the thread is resident inside the component or outside. In particular, we must take into account restrictions due to the fact that the method bodies are executed in *mutual exclusion* wrt. individual objects.

3.1 Balance conditions

We start with auxiliary definitions concerning the parenthetic nature of calls and returns. Starting from an initial configuration, the operational semantics from Section 2.2 assures strict alternation of incoming and outgoing communication and additionally that there is no return without matching prior call.

Definition 2 (Balance). *Let $s \downarrow_n$ be the projection of trace s onto thread n. The balance of a thread n in a sequence s of labels is given by the rules of Table 4, where the dual rules for balanced$^-$ are omitted. We write $\vdash s : balanced_n$ if $\vdash s : balanced^+_n$ or $\vdash s : balanced^-_n$. We call a (not necessarily proper) prefix of a balanced trace* weakly balanced. *We write $\vdash s : wbalanced^+_n$ if the trace is weakly balanced in n, i.e., if the projection of the trace on n is weakly balanced, and if the last label is an incoming communication or if $s \downarrow_n$ is empty; dually for $\vdash s : wbalanced^-_n$. The function pop (on the projection of a trace onto a thread n) is defined as follows:*

$$\frac{}{\vdash \epsilon : \mathit{balanced}^+} \text{ B-EMPTY}^+$$

$$\frac{\vdash s_1 : \mathit{balanced}^+ \qquad \vdash s_2 : \mathit{balanced}^+ \qquad s_1, s_2 \neq \epsilon}{\vdash s_1\ s_2 : \mathit{balanced}^+} \text{ B-II}$$

$$\frac{\vdash s : \mathit{balanced}^-}{\vdash \nu(\Xi).n\langle \mathit{call}\ o_r.l(\vec{v})\rangle!\ s\ \nu(\Xi').n\langle \mathit{return}(v)\rangle? : \mathit{balanced}^+} \text{ B-OI}$$

Table 4. Balance

1. *pop* $s = \bot$, *if* s *is balanced in* n.
2. *pop* $(s_1 a s_2) = s_1 a$ *if* $a = \nu(\Xi).\ n\langle \mathit{call}\ o_r.l(\vec{v})\rangle?$ *and* s_2 *is balanced*^+_n.
3. *pop* $(s_1 a s_2) = s_1 a$ *if* $a = \nu(\Xi).\ n\langle \mathit{call}\ o_r.l(\vec{v})\rangle!$ *and* s_2 *is balanced*^-_n.

We use pop $n\ r$ *for pop* $(r \downarrow_n)$.

Based on a weakly balanced history, we defined the source and target of a communication event at the end of a trace with the help of the function *pop*.

Definition 3 (Sender and receiver). *Let* $r\ a$ *be the non-empty projection of a balanced trace onto the thread* n. *Sender and receiver of label* a *after history* r *are defined by mutual recursion and pattern matching over the following cases:*

$$\begin{aligned}
\mathit{sender}(\nu(\Xi).n\langle \mathit{call}\ o_r.l(\vec{v})\rangle!) &= \odot_n \\
\mathit{sender}(r'\ a'\ \nu(\Xi).n\langle \mathit{call}\ o_r.l(\vec{v})\rangle!) &= \mathit{receiver}(r'\ a') \\
\mathit{sender}(r'\ a'\ \nu(\Xi).n\langle \mathit{return}(v)\rangle!) &= \mathit{receiver}(\mathit{pop}(r'\ a'))
\end{aligned}$$

$$\begin{aligned}
\mathit{receiver}(r\ \nu(\Xi).n\langle \mathit{call}\ o_r.l(\vec{v})\rangle!) &= o_r \\
\mathit{receiver}(r\ \nu(\Xi).n\langle \mathit{return}(v)\rangle!) &= \mathit{sender}(\mathit{pop}(r))
\end{aligned}$$

For $\nu(\Xi)n\langle \mathit{call}\ o_r.l(\vec{v})\rangle?$ *resp.* $\nu(\Xi).n\langle \mathit{return}(v)\rangle?$, *the definition is dual.*

$\Delta, \Sigma \vdash r \rhd a : \Theta, \Sigma$ asserts that after r, the action a is enabled. Input enabledness checks whether, given a sequence of past communication labels, an incoming call is possible in the next step; analogously for output enabledness. To be input enabled, one checks against the last matching communication. If there is no such label, enabledness depends on where the thread started:

Definition 4 (Enabledness). *Given* $\gamma = \nu(\Xi).n\langle \mathit{call}\ o_r.l(\vec{v})\rangle$. *Then* call-enabledness *of* γ *after history* r *and in the contexts* Δ, Σ *and* Θ, Σ *is defined as:*

$$\Delta, \Sigma \vdash r \rhd \gamma? : \Theta, \Sigma \ \text{ if } \ \mathit{pop}\ \ n\ r = \bot \quad \text{and} \quad \Delta \vdash \odot_n \ \text{ or } \ \mathit{pop}\ \ n\ r = r'\gamma'! \tag{1}$$

$$\Delta, \Sigma \vdash r \rhd \gamma! : \Theta, \Sigma \ \text{ if } \ \mathit{pop}\ \ n\ r = \bot \quad \text{and} \quad \Theta \vdash \odot_n \ \text{ or } \ \mathit{pop}\ \ n\ r = r'\gamma'? \tag{2}$$

For return labels $\gamma = \nu(\Xi).n\langle return(v)\rangle$, $\Xi \vdash r \rhd \gamma!$ *abbreviates pop* $n\ r = r'\nu(\Xi').n\langle call\ o_2.l(\vec{v})\rangle?$, *and dually for incoming returns* $\gamma?$.

We further combine enabledness and determining sender and receiver (cf. Definitions 4 and 3) into the notation $\Xi \vdash r \rhd o_s \xrightarrow{a} o_r$.

3.2 Side conditions for monitors

Next we address the restrictions imposed by the fact that the methods are synchronized. We assume in the following that *all* methods are synchronized, unless stated otherwise. We proceed in two stages. The first step in Section 3.2.1 concentrates on individual threads: given the interaction history of a single thread, we present two abstractions, one characterizing situations where the thread *may* hold the lock of a given object, and a second one where, independent of the scheduling, the thread *must* hold the lock. The second step in Section 3.2.2 takes a global view, i.e., considers all threads, to characterize situations in a trace which are (in-)consistent with the fact that objects act as monitors. The formalization is based on a *precedence* or *causal* relation of events of the given trace. This precedence relation formalizes three aspects that regulate the possible orderings of events in a trace:

***mutual exclusion*:** If a thread has taken the lock of a monitor, interactions of other threads with that monitor must either occur *before* the lock is taken, or *after* it has been released again.
***data* dependence:** no value (unless generated new) can be transmitted before it has been received.
***control* dependence:** within a single thread, the events are linearly ordered.

The formalization of mutual exclusion is complicated by the fact that the locks are not taken atomically, i.e., we often do not have *immediate* information when the lock is taken and relinquished. Instead we must work with the may- and must-approximations calculated in Section 3.2.1.

3.2.1 Lock ownership We start by characterizing when, given a history of interaction of a single thread, it *may* own the lock of an object. The "may"-uncertainty is due to the fact that the actual lock manipulation is separated by the corresponding visible interface interaction by some internal i.e., non-observable reduction steps.

Definition 5 (May lock ownership). *Given a sequence s of interactions of a single thread and a component object o, the judgment* $\Xi \vdash s : \Diamond o$ *("after s, the thread of s may own the lock of o.") is given by the rules of Table 5. For environment locks, i.e., when o is an environment object, the definition is dual.*

Rule M-$\Diamond$ states that a strongly balanced tail s_2 can be ignored, lock-wise. The two M-I$\Diamond$-rules deal with incoming calls, depending on the receiver of the communication. If the call concerns the object o in question, the thread may

$$\frac{\vdash s_2 : \mathit{balanced} \qquad s_2 \neq \epsilon \qquad \Xi \vdash s_1 : \Diamond o}{\Xi \vdash s_1\ s_2 : \Diamond o}\ \text{M-}\Diamond$$

$$\frac{\mathit{receiver}(s\gamma_c) = o}{\Xi \vdash s\ \gamma_c? : \Diamond o}\ \text{M-I}\Diamond_1 \qquad \frac{\mathit{receiver}(s\gamma_c) \neq o \quad \Xi \vdash s : \Diamond o}{\Xi \vdash s\ \gamma_c? : \Diamond o}\ \text{M-I}\Diamond_2$$

$$\frac{\Xi \vdash s\ \ : \Diamond o}{\Xi \vdash s\ \gamma_c! : \Diamond o}\ \text{M-O}\Diamond$$

Table 5. Potential lock ownership for Θ-locks

own the lock afterwards. If the receiver is distinct from o (cf. rule M-I$\Diamond_2$), the thread may own the lock of o, if that was the case already before the call. An outgoing call finally does not affect the $\Diamond$-information.

Now to the *definite* knowledge that a thread owns the lock of a given object.

Definition 6 (Must lock ownership). *Given a sequence s of interactions of a single thread and a component object o, the judgment $\Xi \vdash s : \Box o$ ("after s, the thread of s must own the lock of o.") is given by the rules of Table 6. For environment locks, i.e., when o is an environment object, the definition is dual.*

The first rule M-I$\Box_1$ deals with incoming calls. Since the lock is not acquired atomically, an incoming call alone does not guarantee that the thread owns the callee's lock; it potentially owns it according to rule M-I$\Diamond_1$. If however the lock of an object is necessarily owned before the call, the same is true afterwards. Rule M-I$\Box_2$ deals with incoming returns. As for incoming calls, the lock is owned for sure after the communication, if this was true before already. We need to be careful, however. After the return γ_r in question, the thread may continue *internally* i.e., without performing a further interface communication, and this internal reduction may relinquish the lock! This may be the case if the mentioned

$$\frac{\Xi \vdash t : \Box o}{\Xi \vdash t\gamma_c? : \Box o}\ \text{M-I}\Box_1 \qquad \frac{\begin{array}{c}\Xi \vdash t\gamma_r?\gamma_r'! : \Diamond o \\ \Xi \vdash t : \Box o\end{array}}{\Xi \vdash t\gamma_r? : \Box o}\ \text{M-I}\Box_2$$

$$\frac{\Xi \vdash t : \Diamond o}{\Xi \vdash t\gamma_c! : \Box o}\ \text{M-O}\Box_1 \qquad \frac{\Xi \vdash t : \Box o}{\Xi \vdash t\gamma_r! : \Box o}\ \text{M-O}\Box_2$$

Table 6. Necessary lock ownership for Θ-locks

internal reduction includes the very last internal steps of a synchronized method call, before the call actually returns at the interface, re-establishing balance. In other words, after $\gamma_r?$, the component may be in a state where internally, the lock has already been released, only that the fact has not yet been manifest at the interface. This is captured in the premise $\Xi \vdash r\gamma_r?\gamma_r'! : \Diamond o$, i.e., the trace $r\gamma_r?$ is *extended* by one additional outgoing return $\gamma_r'!$, and if the thread *may* have the lock after this extended trace, then it must have the lock after $\gamma_r?$.

The M-O$\Box$-rules cover outgoing communication. Remember that outgoing communication leaves the $\Diamond$-information unchanged. For $\Box$-information, this is different and characteristic of the non-atomic lock-handling: an incoming call is the sign that we *may* have the lock of a component object, but only a following outgoing call is the observable sign that the component *must* have the lock.

We write $\Xi \vdash t : \Box_n o$ for $\Xi \vdash (t \downarrow_n) : \Box o$, and analogously for $\Diamond_n o$.

Lemma 1 (Decidability). *Given a weakly balanced trace t, the relations $\Xi \vdash t : \Diamond_n o$ and $\Xi \vdash t : \Box_n o$ are decidable.*

With decidability at hand we can consider the assertions $\Xi \vdash t : \Diamond_n o$ and $\Xi \vdash t : \Box_n o$ as boolean predicates, and we write $\Xi \vdash t : \neg\Diamond_n o$ for $\Xi \nvdash t : \Diamond_n o$, and analogously for $\Box$.

Lemma 2 ($\Box$ implies $\Diamond$). *Assume a weakly balanced trace t. If $\Xi \vdash t : \Box_n o$ then $\Xi \vdash t : \Diamond_n o$.*

3.2.2 Mutual exclusion So far we concentrated on each thread in isolation. This cannot be the whole story, as mutual exclusion is a global property concerning more than one thread. The formalization is based on a *precedence* relation on the events of a trace. An event is an *occurrence* of a label in a trace, i.e., as usual, events are assumed unique. In the following we do not strictly distinguish (notationally) between labels and events, i.e., we write $\gamma?$ for an event labeled by an incoming communication etc. To formalize the dependencies for mutual exclusion, we need to require that certain events are positioned *before* the lock has been taken, or *after* it has been released. So the following definition picks out relevant events of a trace. In the definition, $\preccurlyeq$ denotes the prefix relation. The $\acute{\Diamond}$-function ("after may") designates the labels *after* the point where the lock may be taken, for a given pair of thread and monitor. The $\grave{\Box}$-function ("before must") picks out the point before a thread enters the monitor.

Definition 7. *Let t be the projection of a weakly balanced trace onto a thread n. Then the set of events $\acute{\Diamond}(t, o)$ is given by:*

$$\acute{\Diamond}(t, o) = \{a \mid \textit{longest prefix} \quad sa \preccurlyeq t \quad \textit{s.t.}\ \Xi \vdash s : \Diamond o\}\ . \tag{3}$$

Furthermore, the set of events $\grave{\Box}(t, o)$ is given as:

$$\begin{aligned}\grave{\Box}(t, o) = \{a_1 \mid\ & \Xi \vdash t : \Box o, \quad \textit{longest prefix}\ sa_1a_2 \preccurlyeq t \quad \textit{s.t.} \\ & \Xi \vdash s : \neg\Diamond o, \quad \Xi \vdash sa_1a_2 : \Box o\ \}\ .\end{aligned} \tag{4}$$

We use the following abbreviations: $\acute{\Diamond}_n(t,o)$ *stands for* $\acute{\Diamond}(t\downarrow_n, o)$ *and* $\acute{\Diamond}_{\neq n}(t,o) = \bigcup_{n' \neq n} \acute{\Diamond}(t \downarrow_{n'}, o)$*, and analogously for* $\grave{\Box}$.

Note that the "set" given by $\acute{\Diamond}$ in Definition 7 contains one element or is empty. The same holds for $\grave{\Box}$.

Based on these auxiliary definitions, we now introduce the three types of dependencies we need to consider. We start with data dependence.

Definition 8 (Data dependence). *Given a trace* r*, reference* o*, and input label* $\gamma?$*, we write* $\vdash_\Theta r : \gamma? \twoheadrightarrow^d o$ *(in words: "o is potentially data-dependent on event/label* $\gamma?$ *of trace r"), if* $o \in names(\gamma)$*, where* $r'\gamma?$ *is a prefix of* r*. When given a tuple* $\vec{o}$ *of names,* $\vdash_\Theta r : \vec{\gamma}? \twoheadrightarrow^d \vec{o}$ *is meant as asserting* $\vdash_\Theta r : \gamma_i? \twoheadrightarrow^d o_i$*, for all* o_i *from* $\vec{o}$ *(for* Δ*, the definitions are applied dually).*

$$\begin{aligned} D_\Theta(r\gamma!) &= \{\vec{\gamma}? \twoheadrightarrow \gamma!\} \quad \textit{where} \vdash_\Theta \vec{\gamma}? \twoheadrightarrow^d \mathit{fn}(\gamma!) \cap \Delta(r) \\ D_\Theta(r\gamma?) &= \{\}\ . \end{aligned} \tag{5}$$

The definition states that, from the perspective of the component, arguments of an outgoing communication must either be generated previously by the component, or must have entered the component from the outside. The complexity of the technical definition is explained as follows. First of all, we calculate the dependence in equation (5) only for object references occurring free in the output label; those that occur under a ν-binder are generated by the component itself, and do not constitute a data dependence. For the same reason we consider only those free object references, which originally have been passed to the component during the history; we denote all ν-bound environment objects in r by $\Delta(r)$ (dually for component objects). Finally, each such object in $\gamma!$ may be potentially data dependent on *more* than one incoming label in the history r. It suffices to add *one* data dependence edge, which is non-deterministically chosen.

Definition 9 (Control dependence). *Given a trace* ra*, where* $n = thread(a)$*, we write* $\vdash r : a' \twoheadrightarrow^c a$*, if* $r\downarrow_n = r'a'$ *for some label* a'*. We write* $C(ra)$ *for* $\{a' \twoheadrightarrow a \mid r \vdash a' \twoheadrightarrow^c a\}$.

Note that the set $C(ra)$ contains one element, i.e., one edge, or is empty.

Definition 10 (Mutual exclusion). *Given a trace* ra *and a component object* o*, the label* a *gives rise to the precedence edges wrt. component locks given by:*

$$\begin{aligned} M_\Theta(r\gamma_c?, o) &= \acute{\Diamond}_{\neq n}(r,o) \twoheadrightarrow \gamma_c? \\ M_\Theta(r\gamma_r?, o) &= \{\} \\ M_\Theta(r\gamma!, o) &= \gamma! \twoheadrightarrow \grave{\Box}_{\neq n}(r,o), \acute{\Diamond}_{\neq n}(r,o) \twoheadrightarrow \grave{\Box}_n(r\gamma!, o) \end{aligned} \tag{6}$$

For environment locks, the definition is dual.

Incoming calls can introduce a dependence with *other* threads n' competing for the concerned lock of the callee. Interactions of a thread n' occurring in the history r *after* n' has applied for the lock (but before $\gamma_c?$) makes evident that n'

succeeded in entering the monitor. Hence the corresponding monitor interactions of n' must have happened before the current incoming call succeeds in entering the monitor. Incoming returns do not introduce new dependencies wrt. Θ-locks (short for component locks), since the return releases the corresponding lock or keeps it, but does not acquire a lock nor competes for it.

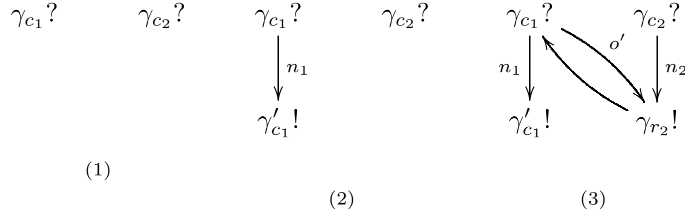
Outgoing communication, however, does introduce dependencies, as they in many cases indicate that a lock definitely is taken or transiently has been taken since the last interaction of that thread. This introduces two types of dependencies. First, if there are other definite lock owners, then the current action $\gamma!$ must precede the monitor interactions of those successful competitors since the outgoing label is a definite sign that the thread of γ has held the lock of o before that step. This explains the edges $\gamma! \rightarrowtriangle \grave{\Box}_{\neq n}(r, o)$ in the definition. Secondly, $\gamma!$ does not only indicate that the thread in question had the lock prior to the step (at least transiently), but can also introduce definite lock ownership after the step (in particular, an outgoing call can introduce must-ownership). Hence, the monitor interactions of all competitors observed in the trace must precede the point, where the current thread n acquires the lock. This explains the dependence $\acute{\Diamond}_{\neq n}(r, o) \rightarrowtriangle \grave{\Box}_n(r\gamma_c!, o)$.

Example 1. Consider the trace $t = \gamma_{c_1}?\ \gamma_{c_2}?\ \gamma'_{c_1}!\ \gamma_{r_2}!$, in expanded form

$$t = (\nu o' {:} c) n_1\langle call\ o.l(o')\rangle?\ n_2\langle call\ o.l()\rangle?\ n_1\langle call\ o'.l()\rangle!\ n_2\langle return(o')\rangle! \quad (7)$$

This trace is impossible because if n_1 were to enter the monitor before n_2, which is required by the data dependency, it implied that n_1 kept the lock and n_2 *could not* enter the monitor. This consequence is independent of the scheduling.

Formally, Definitions 8 – 10 yield the following dependencies, when considering the trace after two, three, or four steps, respectively:



Note that without data dependence from $\gamma_{c_1}?$ to $\gamma_{r_2}!$, the graph is acyclic and the trace possible. Especially, the return $\gamma_{r_2}?$ is possible at the end, even if thread n_1 is guaranteed to hold the lock, since thread n_2 can have performed its monitor interaction before n_1 entered the monitor, only that the return was not yet visible in the trace. □

3.3 Legal traces system

Table 7 specifies *legality* of traces; the rules combine all mentioned conditions, type checking, balance, and in particular restrictions due to monitor behavior.

We use the same conventions and notations as for the operational semantics (cf. Notation 1). The judgments in the derivation system are of the form

$$G_\Delta; \Delta, \Sigma \vdash r \rhd s : trace\, \Theta, \Sigma; G_\Theta \quad \text{resp.} \quad G; \Xi \vdash r \rhd s : trace\ . \tag{8}$$

In comparison to the judgments used in the operational semantics, the judgment from (8) contains a graph G_Θ as representation of *control, data,* and *mutex*-edges wrt. component locks (cf. Section 3.2.2), and dually G_Δ for environment locks. We adapt Notation 1 appropriately, writing G for the pair (G_Θ, G_Δ).

We write $\Xi \vdash t : trace$, if there exists a derivation of $G_\emptyset; \Xi \vdash \epsilon \rhd t : trace$ according to Table 7, where $G_\emptyset$ is the empty dependence graph. We write $\Xi \vdash_\Delta t : trace$, if there exists a derivation of $G_\emptyset; \Xi \vdash \epsilon \rhd t : trace$, where only the *assumption contexts* are checked in the rules but not the commitments, i.e., the premises $\acute{\Xi} \vdash a : ok$ and $\vdash \acute{G} : ok$ *remain* in the rules for incoming communication L-CallI and L-RetI, but for the outgoing communication, the corresponding premises are *omitted* (dually for $\Xi \vdash_\Theta t : trace$).

Now to the rules: As base case, the empty future is always legal, and distinguishing according to the first action a of the trace, the rules check whether a is possible. This check is represented by checking whether the dependencies collected in the pair G are consistent, i.e., that the two graphs are *acyclic*. This is asserted by $\vdash G : ok$. Furthermore, the contexts are updated appropriately, and the rules recur checking the tail of the trace. The update for the dependence graph G_Θ given by the union the graph G_Θ before the step with

$$G_\Theta(ra, o) = M_\Theta(ra, o) \cup C(ra) \cup D_\Theta(ra) \tag{9}$$

where the argument o refers to the monitor relevant in that step, i.e., the monitor introduction potential inconsistencies. The definition for G_Δ is dually.

The rules are completely symmetric wrt. incoming and outgoing communication (and the dual rules omitted). L-CallI for incoming calls works similar

$$\Xi; G \vdash r \rhd \epsilon : trace \quad \text{L-Empty}$$

$$\frac{\begin{array}{c}\Xi \vdash r \rhd o_s \xrightarrow{a} o_r \quad \acute{\Xi} = \Xi + a \quad \acute{\Xi} \vdash a : ok \\ \acute{G}_\Theta = G_\Theta \cup G_\Theta(ra, o_r) \quad \acute{G}_\Delta = G_\Delta \cup G_\Delta(ra, o_s) \quad \vdash \acute{G}_\Delta : ok \\ a = \nu(\Xi').\ n\langle call\ o_r.l(\vec{v})\rangle? \quad \acute{\Xi}; \acute{G} \vdash r\ a \rhd s : trace\end{array}}{\Xi; G \vdash r \rhd a\ s : trace} \ \text{L-CallI}$$

$$\frac{\begin{array}{c}\Xi \vdash r \rhd o_s \xrightarrow{a} o_r \quad \acute{\Xi} = \Xi + a \quad \acute{\Xi} \vdash a : ok \\ \acute{G}_\Theta = G_\Theta \cup G_\Theta(ra, o_r) \quad \acute{G}_\Delta = G_\Delta \cup G_\Delta(ra, o_s) \quad \vdash \acute{G}_\Delta : ok \\ a = \nu(\Xi').\ n\langle return(v)\rangle? \quad \acute{\Xi}; \acute{G} \vdash r\ a \rhd s : trace\end{array}}{\Xi; G \vdash r \rhd a\ s : trace} \ \text{L-RetI}$$

Table 7. Legal traces (dual rules omitted)

to the CallI-rules in the semantics. The premise $\Delta \vdash r \rhd o_s \xrightarrow{a} o_r : \Theta$ checks whether the incoming call a is enabled and determines the sender and receiver at the same time. The receiver o_r, of course, is mentioned directly, but o_s is calculated from the history r. In case of incoming communication, the relevant monitor for G_Θ is the receiver, and for G_Δ, the sender of the step.

Remember from Section 3.1 that the sender given by, e.g., $sender(r\gamma_c?)$ is not (necessarily) the "*real*" sending object (which remains anonymous), but the last environment object the corresponding thread has entered in the past via an interface action. The sender in this sense is exactly the object, whose lock is relevant when updating/checking the dependencies in G_Δ. A consequence of the clean decoupling of component and environment in the assumption/commitment formulation of the legal traces is, that for incoming communication, the update of the graph G_Θ cannot introduce a cycle: incoming communications are checked for legality using the *assumptions,* not the commitments (cf. Lemma 5).

3.4 Soundness of the abstractions

The section contains the basic soundness results of the abstractions.

Lemma 3 (Subject reduction). *$\Xi \vdash C \stackrel{s}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$, then $\acute{\Xi} \vdash \acute{C}$. A fortiori: If $\Xi \vdash n : T$, then $\acute{\Xi} \vdash n : T$.*

Lemma 4 (Soundness of lock ownership).

1. *$\Xi \vdash C \stackrel{t}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ and $\Xi \vdash t : \Box_n o$, then thread n has the lock of o in $\acute{C}$.*
2. *If $\Xi \vdash C \stackrel{t}{\Longrightarrow}$ and $\Xi \vdash t : \Diamond_n o$ and there does not exist an $n' \neq n$ with $\Xi \vdash t : \Box_{n'} o$, then $\Xi \vdash C \stackrel{t}{\Longrightarrow} \acute{\Xi} \vdash \acute{C}$ for some $\acute{\Xi} \vdash \acute{C}$ s.t. the thread n has the lock of o in $\acute{C}$.*

Lemma 5. *If $G; \Xi \vdash r : trace$, and $\Xi \vdash r \rhd o_s \xrightarrow{\gamma?} o_r$, and G_Θ is acyclic, then $G_\Theta + G_\Theta(r\gamma?, o_r)$ is acyclic, as well.*

Lemma 6 (Soundness of abstractions). *Assume $\Xi \vdash C$ and $\Xi \vdash C \stackrel{t}{\Longrightarrow}$. The (1) $\Xi \vdash_\Theta t : trace$ and (2) $\Xi \vdash_\Delta t : trace$ implies $\Xi \vdash t : trace$.*

4 Conclusion

Viswanathan [13] investigates full abstraction in an object calculus with subtyping. The setting is slightly different from the one here, as the paper does not compare a contextual semantics with a denotational one, but a semantics by translation with a direct one. The paper considers neither concurrency nor aliasing. Recently, Jeffrey and Rathke [11] extended their work [10] on trace-based semantics from an object-based setting to a core of *Java*, called *JavaJr*, including classes and subtyping. We plan to extend the language with further features to make it more resembling *Java* or $C^{\#}$. Concerning the concurrency model, one should add thread-coordination using wait- and notify methods. Another

interesting direction for extension concerns the type system, in particular to include *subtyping* and *inheritance*. Another direction is to extend the semantics to a *compositional* one; currently, the semantics is open in that it is defined in the context of an environment. However, general composition of open program fragments is not defined. Concentrating on synchronized methods, this paper relied on an interleaving abstraction of the concurrent semantics. More complex interface behavior is expected when considering more general memory models. See e.g. [12] for a recent semantical study of *Java*'s memory model.

Acknowledgements We thank the anonymous reviewers for their thorough work and their helpful remarks. This work has been financially supported by the NWO/DFG project Mobi-J (RO 1122/9-4) and by the DFG as part of the Transregional collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
2. E. Ábrahám, F. S. de Boer, M. M. Bonsangue, A. Grüner, and M. Steffen. Observability, connectivity, and replay in a sequential calculus of classes. In M. Bosangue, F. S. de Boer, W.-P. de Roever, and S. Graf, editors, *Proceedings of FMCO 2004*, volume 3657 of *LNCS*, pages 296–316. Springer-Verlag, 2005.
3. E. Ábrahám, A. Grüner, and M. Steffen. Abstract interface behavior of object-oriented languages with monitors. Draft technical report, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität zu Kiel, Jan. 2006.
4. E. Ábrahám, A. Grüner, and M. Steffen. Dynamic heap-abstraction for open, object-oriented systems with thread classes (extended abstract). In *Proceedings of CiE'06*, 2006. To appear. A longer version appeared as Technical Report 0601 of the Institute of Computer Science of the University Kiel, January 2006.
5. P. Brinch Hansen. *Operating System Principles*. Prentice Hall, 1973.
6. ECMA International Standardizing Information and Communication Systems. *$C^{\#}$ Language Specification*, 2nd edition, Dec. 2002. Standard ECMA-334.
7. A. D. Gordon and P. D. Hankin. A concurrent object calculus: Reduction and typing. In U. Nestmann and B. C. Pierce, editors, *Proceedings of HLCL '98*, volume 16.3 of *ENTCS*. Elsevier Science Publishers, 1998.
8. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Second edition, 2000.
9. C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.
10. A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings of LICS '02*. IEEE, Computer Society Press, July 2002.
11. A. Jeffrey and J. Rathke. Java Jr.: A fully abstract trace semantics for a core Java language. In M. Sagiv, editor, *Proceedings of ESOP 2005*, volume 3444 of *LNCS*, pages 423–438. Springer-Verlag, 2005.
12. J. Manson, W. Pugh, and S. V. Adve. The Java memory memory. In *Proceedings of POPL '05*. ACM, Jan. 2005.
13. R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proceedings of LICS '98*. IEEE, Computer Society Press, July 1998.